Boids

Giovanni Bruno

September 2025

1 Introduzione

Questo progetto ha lo scopo di simulare il volo di uno stormo di uccelli in volo, emulando il software di AI sviluppato da Craig Reynolds. La simulazione si basa sull'interazione di oggetti, detti boids, che si muovono in un contesto bidimensionale. La dinamica dello stormo è completamente determinata da alcune semplici regole di volo "locali" che ogni boid osserva, cambiando nel tempo la propria velocità e la propria traiettoria. La simulazione permette inoltre di costruire due tipi differenti di boids: le prede ed i predatori. La presenza di predatori rende la simulazione più ricca, perchè nuove regole di volo verranno applicate ad entrambi i tipi di uccelli: i predatori non sentono il bisogno di creare uno stormo coeso, e cercano di catturare le prede; le predecercano di creare uno stormo coeso ed allineato, e fuggono alla caccia di un predatore, quando questo di avvicina.

2 Regole di volo

Le regole di volo a cui obbediscono gli uccelli sono di carattere locale, cioè ogni uccello modifica la propria traiettoria osservando le posizioni e le velocità delle prede e dei predatori vicini. La vicinanza è definita dalla condizione seguente

$$|\vec{x}_{b_i} - \vec{x}_{b_j}| < d \tag{1}$$

Un boid in un certo momento della simulazione considererà come "vicini" i boids che si troveranno in quell'istante ad una distanza inferiore a d. Per rendere la simuazione più realistica, è stato introdotto un angolo di vista, differente per le prede e per i predatori: un boid modifica la propria traiettoria condiderando solamente gli altri boids vicini che si troveranno sul suo campo visivo.

I boids quindi aggiornano le proprie posizioni nel seguente modo:

$$\vec{x}_{\text{fin}} = \vec{x}_{\text{in}} + \vec{v}_{\text{fin}} \cdot \Delta t \tag{2}$$

dove $\mathbf{v}_{\text{fin}}$ è la velocità del boid nel frame corrente, data dalla velocità che il boid aveva nel frame precedente, a cui si somma una variazione di velocità determinata dalle regole di volo, che sono differenti per prede e predatori.

$$\vec{v}_{\text{fin}} = \vec{v}_{\text{in}} + \Delta\vec{v} \tag{3}$$

Per evitare che le regole di volo portassero a velocità troppo elevate, o che un certo boid si possa fermare perchè distante da tutti gli altri boid e non più soggetto ad alcuna interazione, si è deciso di imporre dei limiti di velocità, sia inferiori che superiori, diversi per prede e predatori. Se $\mathbf{v}_{\text{fin}}$ non rispetta i limiti di velocità previsti, verrà moltiplicato per un opportuno fattore di scala. Lo spazio bidimensionale in cui vivono i boids è modellato con condizioni al contorno periodiche, cioè con *wrap-around* (come nel videogioco *Pac-Man*): quando un boid oltrepassa un bordo dell'area di simulazione, ricompare dal lato opposto mantenendo direzione e velocità.

2.1 Prede

Per le prede, la variazione del vettore velocità consiste dei seguenti contributi:

$$\Delta\vec{v} = \vec{v}_{\text{ali}} + \vec{v}_{\text{sep}} + \vec{v}_{\text{coh}} + \vec{v}_{\text{rep}} \tag{4}$$

Il primo contributo è il termine di *alignment*, dato dalla volontà della preda di allinearsi al proprio stormo.

$$\vec{v}_{\text{ali}} = a \left(\frac{1}{n-1} \sum_{j \neq i} \vec{v}_{b_j} - \vec{v}_{b_i} \right) \tag{5}$$

Dove la sommatoria è estesa a tutte le (altre) prede nel proprio campo visivo distanti una lunghezza minore di d. Il secondo contributo è il termine di *separation*, dato dalla volontà della preda di allontanarsi dai boids per evitare urti in volo.

$$\vec{v}_{\text{sep}} = -s \sum_{\substack{j \neq i \\ \|\vec{x}_{b_i} - \vec{x}_{b_j}\| < d_s}} \left(\vec{x}_{b_j} - \vec{x}_{b_i} \right) \tag{6}$$

La sommatoria è estesa a tutti i boids (sia prede che predatori) che distano dalla preda considerata una lunghezza inferiore a ds_{prey}. Il terzo contributo è dato dal termine di *cohesion*, che induce la preda ad avvicinarsi al centro di massa dei propri vicini

$$\vec{x}_{CM} = \frac{1}{n-1} \sum_{j \neq i} \vec{x}_{b_j} \tag{7}$$

$$\vec{v}_{\text{coh}} = c \left(\vec{x}_{CM} - \vec{x}_{b_i} \right) \tag{8}$$

Anche qui la sommatoria è estesa a tutte le (altre) prede nel proprio campo visivo distanti una lunghezza minore di d. Il quarto contributo è il termine di *repulsion* e permette alla preda di fuggire dai predatori vicini. La regola è analoga a quella di *separation*, ma adesso consideriamo solamente i predatori presenti nel cerchio di raggio ds_{prey}. nel campo visivo della preda in questione, ed usiamo un coefficiente differente da s, chiamato r.

2.2 Predatori

Per i predatori, la variazione del vettore velocità consiste dei seguenti contributi:

$$\Delta\vec{v} = \vec{v}_{\text{sep}} + \vec{v}_{\text{cha}} \tag{9}$$

Il primo contributo è il termine di *separation*, identico a quello presente nelle prede. Il secondo contributo è il termine di *chase* e permette al predatore di inseguire le prede vicine (presenti nel cerchio di raggio ds_{predator}, nel suo campo visivo). La regola è analoga al termine di cohesion, usando stavolta il coefficiente ch: al predatore è sommata una velocità che punta nella direzione del centro di massa delle prede vicine.

3 Struttura del programma

Il programma è organizzato come segue:

- cinque file di intestazione, contenuti nella cartella `/include`;
- i corrispondenti file sorgente, contenuti nella cartella `/src`;
- nella cartella `/src` sono presenti anche:
  - `main.cpp`, contenente la funzione `main()`;
  - `test.cpp`, che include le unit test implementate utilizzando la libreria `doctest`.

Le classi, le struct e le funzioni implementate sono state ripartite nel progetto tra i seguenti namespace: `point, boid, flock, graphics, statistics`

3.1 `point.hpp, point.cpp`

In questa parte del programma è stata dichiarata e definita la classe `Point`. Essa presenta due membri privati, `x_` e `y_`, e i seguenti metodi pubblici:

- un costruttore parametrico esplicito;
- i getters `getX` e `getY`;

- i setters `setX` e `setY`;
- un metodo `distance`, che restituisce la norma euclidea del punto rispetto all'origine;
- un overload di `distance` che prende come argomento un altro `Point` e restituisce la norma euclidea del vettore differenza, utile per calcolare distanze e velocità relative;
- l'operatore `+=`, che somma componente per componente un altro `Point` al `Point` su cui è stato chiamato.

Nel namespace `point` sono inoltre definite alcune funzioni libere:

- `operator+`: restituisce il `Point` somma di due `Point`;
- `operator-`: restituisce il `Point` differenza di due `Point`;
- `operator*`: moltiplica un `Point` per uno scalare `double`;
- `operator/`: divide un `Point` per uno scalare `double`;
- `operator==`: verifica l'uguaglianza tra due `Point`, confrontando le componenti;
- `relativePosition`: calcola il raggio vettore relativo del secondo `Point` rispetto al primo, considerando la geometria toroidale;
- `toroidalDistance`: calcola la distanza euclidea tra due `Point` tenendo conto della geometria toroidale.

3.2 `boid.hpp, boid.cpp`

In questi file sono dichiarate e definite le classi `Boid`, `Prey` e `Predator`.

3.2.1 `boid`

`Boid` è una classe base astratta, che rappresenta genericamente un agente (boid). Contiene i seguenti membri protetti:

- `position_`: un `Point` che rappresenta la posizione del boid nello spazio;
- `velocity_`: un `Point` che rappresenta la velocità del boid.

Sono inoltre presenti i seguenti metodi pubblici:

- un costruttore parametrico e il distruttore virtuale;
- i metodi `getPosition` e `getVelocity`, che restituiscono i valori dei membri privati;
- il metodo `setBoid`, che permette di aggiornare simultaneamente posizione e velocità;
- il metodo `angle`, che calcola l'angolo tra la direzione di moto del boid e il raggio vettore relativo rispetto a un altro boid;
- il metodo `separation`, che implementa la regola di separazione: calcola un vettore di spostamento che allontana il boid dai vicini che si trovano entro una certa distanza ds;
- il metodo virtuale puro `clamp`, che forza la velocità del boid a rimanere compresa tra una velocità minima e una massima.

La presenza del metodo virtuale puro rende `Boid` una classe astratta e abilita l'uso del **polimorfismo**: un puntatore a `Boid` può riferirsi sia a un `Prey` sia a un `Predator`, consentendo di trattare in modo uniforme insiemi eterogenei di agenti.

3.2.2 Classe `Prey`

`Prey` è una classe derivata finale di `Boid`, che rappresenta un agente di tipo "preda". Oltre ai metodi ereditati da `Boid`, implementa:

- `alignment`: calcola la variazione di velocità necessaria per allinearsi alla velocità media dei vicini;
- `cohesion`: calcola il vettore che porta il boid verso il baricentro dei boid vicini;
- `repulsion`: calcola il vettore di fuga rispetto ai predatori vicini;
- l'override del metodo `clamp`, che impone vincoli alla velocità.

3.2.3 Classe `Predator`

`Predator` è una classe derivata finale di `Boid`, che rappresenta un agente di tipo "predatore". Definisce:

- `chase`: calcola un vettore che orienta il predatore verso le prede vicine, implementando il comportamento di inseguimento;
- l'override di `clamp`, analogo a quello di `Prey`.

Ereditarietà e Polimorfismo Questa gerarchia di classi permette di utilizzare un contenitore (`std::vector<std::shared_ptr<Boid>>`) per memorizzare sia prede che predatori. Grazie al polimorfismo, è possibile chiamare metodi virtuali come `clamp` su oggetti di tipo `Prey` e `Predator` attraverso un'interfaccia comune, senza conoscere il tipo dinamico esatto dell'oggetto.

3.3 `flock.hpp, flock.cpp`

La classe `Flock` rappresenta l'insieme degli agenti (prede e predatori) nella simulazione e gestisce la creazione (tramite la generazione casuale di due vettori di oggetti derivati `Prey` e `Predator`), il calcolo dei vicini, l'aggiornamento delle posizioni e delle velocità e le statistiche della popolazione di boid.

Membri privati principali

- **Parametri numerici e generatori casuali:**
  - `std::mt19937 mt_` – generatore di numeri pseudocasuali basato sull'algoritmo dimarsenn twister.
  - `n_prey_`, `n_predators_` – numero di prede e predatori.
- **Contenitori dei boid:**
  - `prey_flock_`, `predator_flock_` – vettori di smart pointer ai rispettivi boid.
- **Parametri di simulazione:**
  - `FlightParameters flight_parameters_` – struct di parametri: separazione, allineamento, coesione, repulsione, inseguimento.
  - `SpeedLimits speed_limits_` – struct dei limiti di velocità per prede e predatori.
- **Parametri statici:**
  - `d_`, `prey_ds_`, `predator_ds_` – raggi di vicinanza e separazione.
  - `prey_sight_angle_`, `predator_sight_angle_` – angoli di vista degli agenti.

Costruttori

- `Flock(std::size_t n_prey, std::size_t n_predators)` – costruisce un oggetto Flock vuoto con il numero di prede e predatori indicato, usando valori di default per parametri di volo e limiti di velocità.
- `Flock(const std::vector<std::shared_ptr<boid::Prey>> &prey, const std::vector<std::shared_ptr<boid::Predator>> &predators, const SpeedLimits &speed_limits)` – costruisce un oggetto Flock con boid già creati e limiti di velocità personalizzati.

Metodi pubblici principali

Getters:

- `getPreyNum()`, `getPredatorsNum()`, `getFlockSize()` – restituiscono il numero di agenti.
- `getPreyFlock()`, `getPredatorFlock()` – restituiscono i vettori dei boid.
- `getFlightParameters()`, `getSpeedLimits()` – restituiscono i parametri di simulazione.
- `getDistanceParameters()` – restituisce i raggi statici `d_`, `prey_ds_`, `predator_ds_`.

Setters e generazione:

- `setFlockSize()` – permette di inserire il numero di prede e predatori, vengono usati valori di default (200 prede e 5 predatori) se l'input è non valido (è stato deciso di dare un numero massimo di prede pari a 1000 e di predatori pari a 100).
- `setFlightParameters()` – consente di personalizzare i parametri di separazione, allineamento, coesione, repulsione e inseguimento, oppure usare i valori di default.
- `generateBoids()` – genera boid (pseudo)casuali con posizione e velocità iniziali distribuite uniformemente, utilizzando oggetti della classe `std::uniform_real_distribution<>`, che trasformano i numeri interi prodotti dal generatore `mt_` della classe `std::mt19937` in valori reali nell'intervallo desiderato. Le posizioni vengono generate uniformemente nel rettangolo di lati `window_width` e `window_height`, mentre le velocità vengono generate scegliendo in modo uniforme la lunghezza del vettore velocità (la magnitude) in un intervallo pari a [2., 5.] e scegliendo in modo uniforme la direzione, generando angoli nell'intervallo $[0,2\pi]$

Comportamenti e aggiornamento:

- `nearPrey(std::size_t i, bool is_prey)` – restituisce i boid di tipo preda entro distanza `d_` e angolo di vista.
- `nearPredators(std::size_t i, bool is_prey)` – restituisce i boid di tipo predatore vicini.
- `updateBoid(std::size_t i, bool is_prey, double dt)` – aggiorna posizione e velocità di un singolo boid secondo le regole di separazione, allineamento, coesione, repulsione e inseguimento.
- `updateFlock(double dt)` – aggiorna tutti i boid della simulazione.

Statistiche:

- `statistics()` – calcola la distanza media tra le prede, la deviazione standard delle distanze, la velocità media e la deviazione standard delle velocità.

3.4 `statistics.hpp, statistics.cpp`

La struct `Statistics` incapsula le grandezze statistiche della simulazione dei boid, principalmente per analizzare le prede. Non contiene metodi di calcolo: tutte le statistiche vengono calcolate dalla classe `Flock` e passate al costruttore parametrico.

Membri principali

- `mean_distance` – distanza media tra le prede.
- `dev_distance` – deviazione standard delle distanze tra le prede.
- `mean_velocity` – velocità media delle prede.
- `dev_velocity` – deviazione standard delle velocità delle prede.

Costruttori

- `Statistics()` – costruttore di default, inizializza tutte le statistiche a 0.
- `Statistics(double mean_dist, double dev_dist, double mean_vel, double dev_vel)` – costruttore parametrico, permette di inizializzare le statistiche con valori calcolati.

3.5 graphics.hpp, graphics.cpp

La namespace `graphics` gestisce la rappresentazione visiva dei boid tramite la libreria SFML.

Costanti principali

- `window_width, window_height` – dimensioni della finestra grafica.

Struttura

- `Style` – una struct che contiene parametri visivi della simulazione:
  - dimensioni dei boid (`prey_size`, `predator_size`)
  - spessore del bordo (`stroke`)
  - colori dei boid e dello sfondo (`prey_fill`, `predator_fill`, `prey_outline`, `predator_outline`, `backgrount`)

Funzioni principali

- `loadBackground(filename)` – carica un'immagine da usare come sfondo della finestra grafica. In particolare:
  - tenta di caricare l'immagine dal file `filename` usando `backgroundTexture.loadFromFile`. Se fallisce, restituisce `false`.
  - associa la texture a `backgroundSprite`, lo sprite effettivamente disegnato nella finestra.
  - calcola i fattori di scala orizzontali e verticali in modo che l'immagine riempia tutta la finestra indipendentemente dalle dimensioni originali.
  - ritorna `true` se il caricamento ha avuto successo,
- `clearBackground()` – pulisce lo sfondo della finestra con il colore specificato in `Style`.
- `makeWindow(width, height, title)` – crea e restituisce una finestra SFML con impostazioni grafiche, tra cui l'anti-aliasing.
- `angleDegFromVelocity(vx, vy)` – calcola l'angolo di rotazione di un boid a partire dalla velocità.
- `makeBoidTriangle(size, stroke, fill, outline)` – genera una forma triangolare della classe `sf::ConvexShape` per rappresentare un boid.
- `drawBoid(window, x, y, vx, vy, style, is_prey)` – disegna un singolo boid nella finestra, orientato secondo la velocità e colorato in base al tipo.
- `drawFrame(window, flock, style)` – aggiorna la finestra grafica per un frame:
  - Disegna lo sfondo: se la texture è caricata, usa `backgroundSprite`, altrimenti `style.background`
  - Disegna tutte le prede con `drawBoid`, orientate secondo la velocità e colorate secondo `Style`.
  - Disegna tutti i predatori allo stesso modo, ma con colori e dimensioni differenti.

3.6 main.cpp

Il file `main.cpp` contiene il punto di ingresso del programma e gestisce il ciclo principale della simulazione.

- crea un oggetto `Flock` vuoto e richiama le funzioni `setFlockSize()`, `setFlightParameters()` e `generateBoids()` per determinare il numero di agenti, i parametri di simulazione e generare posizioni e velocità iniziali.
- crea la finestra SFML con `graphics::makeWindow()` e inizializza lo `Style` per disegnare boid e sfondo.
- tenta di caricare uno sfondo tramite `graphics::loadBackground()`, in caso di errore usa il colore di default. Carica un font per visualizzare le statistiche a schermo.

- crea un pannello semitrasparente (`sf::RectangleShape`) e un oggetto `sf::Text` per mostrare le statistiche.
- **Ciclo principale while:**
  - Gestisce gli eventi (chiusura finestra, tasto ESC).
  - Aggiorna lo stato del flock con `updateFlock(dt)` in base al tempo trascorso (`sf::Clock`).
  - Disegna sfondo, boid e rettangolo delle statistiche chiamando `graphics::drawFrame()` e visualizza le statistiche se ci sono abbastanza prede.
  - Mostra il frame aggiornato con `window->display()`.

4 Compilazione, esecuzione e testing

La compilazione del progetto avviene tramite `CMake`, uno strumento per la generazione di *build system* che facilita la gestione delle dipendenze del progetto e consente di creare un unico eseguibile finale, rendendo il processo di configurazione e compilazione più flessibile e portabile.

Per visualizzare l'interfaccia grafica, è necessario installare la libreria grafica `SFML` (Simple and Fast Multimedia Library), che fornisce strumenti semplici ed efficienti per la gestione della grafica.

4.1 Compilazione in `Release` mode

Per compilare il progetto in modalità `Release`:

```
cmake -S ./ -B build/release -DBUILD_TESTING=True -DCMAKE_BUILD_TYPE=Release
cmake --build build/release
```

Vengono così costruiti gli eseguibili `Boids` e `Boids.t` nella cartella `/build/release/`.
Per eseguire il programma principale:

```
build/release/Boids
```

Per eseguire i test:

```
build/release/Boids.t
```

4.2 Compilazione in `Debug` mode

Per compilare il progetto in modalità `Debug`:

```
cmake -S ./ -B build/debug -DBUILD_TESTING=True -DCMAKE_BUILD_TYPE=Debug
cmake --build build/debug
```

Eseguibili generati:

```
build/debug/Boids
build/debug/Boids.t
```

4.3 Disabilitare la compilazione dei test

Per evitare di compilare i test, è possibile configurare `CMake` con la flag:

```
-DBUILD_TESTING=False
```

5 Input e output della simulazione

Al momento dell'esecuzione, viene chiesto all'utente di inserire in input da terminale il numero di prede e il numero di predatori che si intende simulare. A seguire, viene data la possibilità di scegliere i parametri di separazione, allineamento e coesione. In caso di input non valido verranno applicati i valori di default. Terminata l'acquisizione dei parametri, essi non potranno essere più modificati nel corso della simulazione. Quindi apparirà la finestra grafica e il rettangolo semitrasparente delle grandezze statistiche in alto a sinistra.

6 Interpretazione qualitativa dei risultati

Osservando la rappresentazione grafica della simulazione con i parametri di default, si nota che i boid tendono spontaneamente a raggrupparsi in piccoli sotto-stormi, che nel tempo convergono verso un unico grande stormo. L'allineamento delle velocità è evidente: gli agenti si dispongono in modo coerente e formano pattern di movimento collettivo, simulando un comportamento simile a quello osservato in natura.

Quando un predatore entra nel raggio visivo di una o più prede, l'effetto è fortemente dispersivo: le prede accelerano e si allontanano dalla zona di pericolo. Questo porta a un aumento temporaneo della velocità media e della sua deviazione standard, in particolare quando il numero di prede è ridotto, poiché ogni predatore ha un'influenza maggiore sulla dinamica complessiva.

Variando il numero di prede, si osserva che stormi più numerosi hanno una dinamica più coerente e stabile: la coesione tende a dominare e le deviazioni individuali si smorzano più rapidamente. All'aumentare del numero di predatori, la dispersione diventa più frequente e il sistema tende a rimanere meno compatto.

Infine, i parametri influenzano in modo significativo la simulazione: un s elevato accentua l'effetto di repulsione, producendo stormi più sparpagliati; un a maggiore rende il movimento più coordinato e liscio; un c alto favorisce il collasso verso un centro comune, accelerando la formazione di uno stormo unico.

L'insieme di questi comportamenti è coerente con le aspettative per un modello di boid biologicamente plausibile.

7 Test

Il file dei test, denominato `test.cpp`, si trova nella cartella `/src`. I test sono stati implementati utilizzando la libreria **DOCTEST**, che consente di scrivere i casi di test direttamente nel codice sorgente in modo compatto e leggibile. La libreria genera autonomamente una funzione `main()` per l'esecuzione dei test, a condizione che, all'inizio del file sorgente, venga inclusa la seguente direttiva:

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
```

I test sono stati progettati per verificare la correttezza dell'implementazione, concentrandosi principalmente su casi semplici e controllati, per accertare che le funzionalità di base si comportino come atteso.

Anche la componente grafica ha avuto un ruolo importante durante la fase di debugging: la visualizzazione interattiva ha infatti permesso di individuare più facilmente comportamenti anomali o errori logici, che sarebbero stati meno immediati da rilevare tramite un'analisi esclusivamente testuale.

8 Link

Per la gestione di questo progetto ho usato una repository *GitHub* di cui riporto il link:

`https://github.com/giovi-chan/Boids`